**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**КЕМЕРОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Институт Цифры**

**ОТЧЕТ**

**О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ №6**

по дисциплине «Введение в нейронные сети»

студента 2 курса

**Лободы Максима Алексеевича**

Направление 02.03.02– «Фундаментальная информатика и информационные технологии»

Преподаватель:

ст. пр. Зимин А. И.

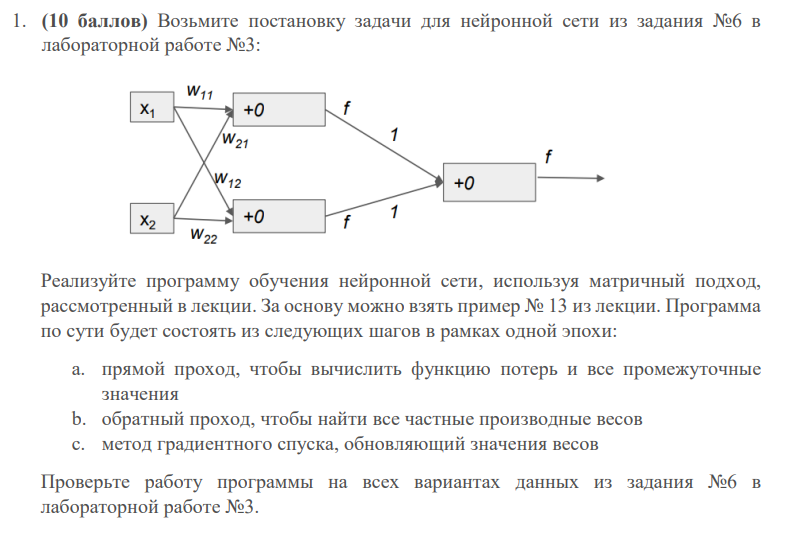
Работа защищена

«\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_»

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_2023 г.

Кемерово 2023

**Основная часть**



Код программы

from \_\_future\_\_ import annotations  
  
import numpy as np  
from numpy import ndarray  
from sklearn.datasets import make\_regression  
from sklearn.linear\_model import LinearRegression  
  
  
def mae(preds: ndarray, actuals: ndarray):  
 return np.mean(np.abs(preds - actuals))  
  
  
def rmse(preds: ndarray, actuals: ndarray):  
 return np.sqrt(np.mean(np.power(preds - actuals, 2)))  
  
  
def init\_weights(input\_size: int, hidden\_size: int) -> dict[str, ndarray]:  
 weights: dict[str, ndarray] = {}  
 weights['W1'] = np.random.randn(input\_size, hidden\_size)  
 weights['B1'] = np.random.randn(1, hidden\_size)  
 weights['W2'] = np.random.randn(hidden\_size, 1)  
 weights['B2'] = np.random.randn(1, 1)  
 return weights  
  
  
Batch = tuple[ndarray, ndarray]  
  
  
def generate\_batch(X: ndarray, y: ndarray, start: int = 0, batch\_size: int = 10) -> Batch:  
 assert X.ndim == y.ndim == 2, "X and Y must be 2 dimensional"  
 if start + batch\_size > X.shape[0]:  
 batch\_size = X.shape[0] - start  
 X\_batch, y\_batch = X[start:start + batch\_size], y[start:start + batch\_size]  
 return X\_batch, y\_batch  
  
  
def permute\_data(X: ndarray, y: ndarray):  
 perm = np.random.permutation(X.shape[0])  
 return X[perm], y[perm]  
  
  
def sigmoid(x: ndarray) -> ndarray:  
 return 1 / (1 + np.exp(-1.0 \* x))  
  
  
def sigmoid\_der(x: ndarray) -> ndarray:  
 return sigmoid(x) \* (1 - sigmoid(x))  
  
  
def relu(data: ndarray) -> ndarray:  
 c = data.copy()  
 c[c <= 0] = 0  
 return c  
  
  
def relu\_der(data: ndarray) -> ndarray:  
 c = data.copy()  
 c[c <= 0] = 0  
 c[c > 0] = 1  
 return c  
  
  
def activation(data: ndarray) -> ndarray:  
 return sigmoid(data)  
  
  
def activation\_der(data: ndarray) -> ndarray:  
 return sigmoid\_der(data)  
  
  
def forward\_loss(X: ndarray, y: ndarray, weights: dict[str, ndarray]) -> tuple[ndarray, dict[str, ndarray]]:  
 M1 = np.dot(X, weights['W1'])  
 N1 = M1 + weights['B1']  
 O1 = activation(N1)  
 M2 = np.dot(O1, weights['W2'])  
 P = M2 + weights['B2']  
 loss = np.mean(np.power(y - P, 2))  
 forward\_info: dict[str, ndarray] = {}  
 forward\_info['X'] = X  
 forward\_info['M1'] = M1  
 forward\_info['N1'] = N1  
 forward\_info['O1'] = O1  
 forward\_info['M2'] = M2  
 forward\_info['P'] = P  
 forward\_info['y'] = y  
 return loss, forward\_info  
  
  
def loss\_gradients(forward\_info: dict[str, ndarray], weights: dict[str, ndarray]) -> dict[str, ndarray]:  
 dLdP = -2 \* (forward\_info['y'] - forward\_info['P'])  
 dPdM2 = np.ones\_like(forward\_info['M2'])  
 dLdM2 = dLdP \* dPdM2  
 dPdB2 = np.ones\_like(weights['B2'])  
 dLdB2 = (dLdP \* dPdB2).sum(axis=0)  
 dM2dW2 = np.transpose(forward\_info['O1'], (1, 0))  
 dLdW2 = np.dot(dM2dW2, dLdP)  
 dM2dO1 = np.transpose(weights['W2'], (1, 0))  
 dLdO1 = np.dot(dLdM2, dM2dO1)  
 dO1dN1 = activation\_der(forward\_info['N1'])  
 dLdN1 = dLdO1 \* dO1dN1  
 dN1dB1 = np.ones\_like(weights['B1'])  
 dN1dM1 = np.ones\_like(forward\_info['M1'])  
 dLdB1 = (dLdN1 \* dN1dB1).sum(axis=0)  
 dLdM1 = dLdN1 \* dN1dM1  
 dM1dW1 = np.transpose(forward\_info['X'], (1, 0))  
 dLdW1 = np.dot(dM1dW1, dLdM1)  
 loss\_gradients: dict[str, ndarray] = {}  
 loss\_gradients['W2'] = dLdW2  
 loss\_gradients['B2'] = dLdB2.sum(axis=0)  
 loss\_gradients['W1'] = dLdW1  
 loss\_gradients['B1'] = dLdB1.sum(axis=0)  
 return loss\_gradients  
  
  
def train(X: ndarray, y: ndarray, n\_iter: int = 1000, learning\_rate: float = 0.1, batch\_size: int = 100,  
 return\_losses: bool = False, return\_weights: bool = False, seed: int = 1) -> tuple[list[ndarray], dict[str, ndarray]] | None:  
 if seed:  
 np.random.seed(seed)  
 start = 0  
 weights = init\_weights(X.shape[1], 2)  
 R: dict[str, ndarray] = {}  
 R['W1'] = np.zeros\_like(weights['W1'])  
 R['B1'] = np.zeros\_like(weights['B1'])  
 R['W2'] = np.zeros\_like(weights['W2'])  
 R['B2'] = np.zeros\_like(weights['B2'])  
 M: dict[str, ndarray] = {}  
 M['W1'] = np.zeros\_like(weights['W1'])  
 M['B1'] = np.zeros\_like(weights['B1'])  
 M['W2'] = np.zeros\_like(weights['W2'])  
 M['B2'] = np.zeros\_like(weights['B2'])  
 e0 = 1e-6  
 b1 = 0.8  
 b2 = 0.88  
 X, y = permute\_data(X, y)  
 if return\_losses:  
 losses = []  
 for i in range(n\_iter):  
 if start >= X.shape[0]:  
 X, y = permute\_data(X, y)  
 start = 0  
 X\_batch, y\_batch = generate\_batch(X, y, start, batch\_size)  
 start += batch\_size  
 loss, forward\_info = forward\_loss(X\_batch, y\_batch, weights)  
 if return\_losses:  
 losses.append(loss)  
 loss\_grads = loss\_gradients(forward\_info, weights)  
 for key in weights.keys():  
 M[key] = b1 \* M[key] + (1 - b1) \* loss\_grads[key]  
 R[key] = b2 \* R[key] + (1 - b2) \* np.power(loss\_grads[key], 2)  
 weights[key] -= (M[key] / (np.sqrt(R[key] + e0))) \* learning\_rate  
 if return\_weights:  
 return losses, weights  
 return None  
  
  
def predict(X: ndarray, weights: dict[str, ndarray]) -> ndarray:  
 M1 = np.dot(X, weights['W1'])  
 N1 = M1 + weights['B1']  
 O1 = activation(N1)  
 M2 = np.dot(O1, weights['W2'])  
 P = M2 + weights['B2']  
 return P  
  
  
X, y, c = make\_regression(n\_samples=100, n\_features=7, n\_informative=2, noise=3, coef=True)  
y = y.reshape((-1, 1))  
X\_train, y\_train = X[:70], y[:70]  
X\_test, y\_test = X[70:], y[70:]  
X\_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  
y\_train = np.array([[0], [0], [0], [1]])  
X\_test, y\_test = X\_train, y\_train  
losses, weights = train(X\_train, y\_train,  
 n\_iter=1000,  
 learning\_rate=0.1,  
 batch\_size=1,  
 return\_losses=True,  
 return\_weights=True)  
print(\*losses[-5:])  
preds = predict(X\_train, weights)  
print(preds)  
preds = predict(X\_test, weights)  
print(preds)  
print("Mean absolute error:", round(mae(preds, y\_test), 4), "\nRoot mean squared error:",  
 round(rmse(preds, y\_test), 4))  
lr = LinearRegression(fit\_intercept=True)  
lr.fit(X\_train, y\_train)  
preds = lr.predict(X\_test)  
print(preds)  
print("Mean absolute error:", round(mae(preds, y\_test), 4), "\nRoot mean squared error:",  
 round(rmse(preds, y\_test), 4))  
print(np.round(weights['W1'].reshape(-1), 4))  
print(np.round(weights['W2'].reshape(-1), 4))  
print(np.round(lr.coef\_, 4))  
print(np.round(weights['B1'], 4))  
print(np.round(weights['B2'], 4))  
print(np.round(lr.intercept\_, 4))  
print("True coef:", c)

Результат выполнения

